Ryan Howard

8/11/2024

CS 300

Project One Pseudocode

Menu in main:

Create a variable to hold user choice and initialize to 0

While user input does not equal 9

Display menu options

Switch with user choice

Case 1 calls function to load courses

Case 2 calls sorted print function for specified data structure

Case 3 calls the search function for specified data structure and prints data found

Case 9 terminates program displaying a goodbye message

End while loop

Alphanumeric print function:

VECTOR:

Function printSort(vector, begin, end):

Call to quicksort function with course vector and begin and end point passed

For every index display course information

Break

Function partition(vector courses, int begin, int end):

Create an integer variable to hold the low index and set it to begin

Create an integer variable to hold the high index and set it to end

Create an integer variable to hold the midpoint of low and high indexes

Create a string variable to hold the bid title for comparisons

Create a Boolean variable to control the loop

While done is false

While the low index title is less than the pivot string

Increment the low index

While the high index title is greater than the pivot string

Decrement the high index

If the low index is greater than or equal to the high index

Set done to true

Else

Call the swap function with arguments low index and high index

Increment low index

Increment high index

End the outer while loop

Return the high index

Function quicksort(vector courses, int begin, int end):

Create an integer variable to hold mid point and set it to zero

If begin is greater than or equal to end

Return

Set mid point variable to a function call of partition with the passed parameters

Call the quicksort function recursively with arguments: course, begin, mid

Call the quicksort function recursively with arguments: course, mid+1, end

TREE:

Function inOrder(node):

If the node is not null

Recursively call inOrder with the left child

Output the course information of the current node

Recursively call inOrder with the right child

HASHTABLE:

Function printInOrder():

Create a temporary vector to hold course objects

For each bucket from beginning to end

If the key does not equal UINT\_MAX

Add node to vector

Create a node pointer initialized to the current next node

While the node pointer is not null

Add node to vector

Set node pointer to the next node

End while leep

End for loop

Call a quicksort function on the vector

For every index display course information

RUNTIME ANALYSIS OF READING FILES AND CREATING COURSE OBJECTS:

|  |  |  |  |
| --- | --- | --- | --- |
| **Vector** | **Cost Per Line** | **Times Executed** | **Total Cost** |
| Create a vector | 1 | 1 | 1 |
| Create temp string variable | 1 | 1 | 1 |
| Open file | 1 | 1 | 1 |
| While file has not ended loop (including creating course object, and adding to vector) | 7 | n | 7n |
| While line is not at end loop | 1 | n | n |
| **COST** | | | 8n + 3 |
| **RUNTIME** | | | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Hashtable** | **Cost Per Line** | **Times Executed** | **Total Cost** |
| Create a temp string variable | 1 | 1 | 1 |
| Open file | 1 | 1 | 1 |
| While file has not ended (including creating course object) | 6 | n | 6n |
| While line has not ended | 1 | n | n |
| Insert course | 4 | 1 | 4 |
| While next pointer is not null | 1 | n | n |
| **COST** | | | 8n + 6 |
| **RUNTIME** | | | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Tree** | **Cost Per Line** | **Times Executed** | **Total Cost** |
| Create a temp string variable | 1 | 1 | 1 |
| Open file | 1 | 1 | 1 |
| While file has not ended (including creating course object) | 6 | n | 6n |
| While line has not ended | 1 | n | n |
| Insert course call addNode | 3 | n | 3n |
| **COST** | | | 10n + 2 |
| **RUNTIME** | | | O(n) |

It is clear that all three of these data structures have very similar runtimes. Based on runtime of loading files and creating objects the vector structure appears to be the fastest. A disadvantage to using a vector is the searching of a course requires checking each index until the course is found.

Hash tables are able to search a list faster than vectors, but have a slightly slower runtime while creating courses and loading the file. The major disadvantage of hash tables in this program is that in order to print an alphanumeric list the data must be ordered into another structure before printing. This is costly and time consuming.

Binary search trees are faster than vectors for sorting, however they also have the slowest implementation time in regards to loading the file and creating courses. Binary search trees also allow for fast searching when compared to vectors.

I will be using a binary search tree in my project. All of the structures have a Big O of O(n). Hash tables are inefficient for the program due to the lack of sorting capabilities. A vector may require slightly less time than a binary search tree in the creation, however a binary search tree is much more efficient in its searching capabilities.